DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 7
Bases de Datos noSQL

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

UD7 – Bases de Datos noSQL

ÍNDICE

7. Conexión a MongoDB desde Java
- 7.1 Documentación y driver
- 7.2 Establecer la conexión
- 7.3 Obtener datos
- 7.4 Insertar datos
- 7.5 Actualizar datos
- 7.6 Eliminar datos
- 7.7 Aplicando filtros y orden

7. Conexión a MongoDB desde Java

7.1 Documentación y driver

Para poder utilizar una base de datos en MongoDB, al igual que ocurre con otras BD como MySQL, debemos instalar el driver necesario para poder realizar las operaciones. Independientemente del IDE que estemos utilizando para programar en JAVA, deberemos descargar los ficheros jar necesarios y añadirlos a nuestros proyectos.

El driver de MongoDB para JAVA en su última versión requiere las siguientes dependencias jar que debemos satisfacer.

- **`mongodb-driver-core-X.X.X.jar`**
- **`mongodb-driver-X.X.X.jar`**
- **`bson-X.X.X.jar`**

También podemos añadir los Javadoc para tener la documentación relacionada con las APIs que acabamos de añadir.

- **`mongodb-driver-core-X.X.X-javadoc.jar`**
- **`mongodb-driver-X.X.X-javadoc.jar`**
- **`bson-X.X.X-javadoc.jar`**

Consultar la versión del driver según versión del servidor de MongoDB
https://docs.mongodb.com/ecosystem/drivers/java/

Para descargarlos podemos utilizar estos enlaces.

MongoDB Driver - Download
https://repo.maven.apache.org/maven2/org/mongodb/mongodb-driver/

BSON - Download
https://oss.sonatype.org/content/repositories/releases/org/mongodb/bson/

A continuación podéis encontrar documentación sobre el driver.

API MongoDB JAVA
https://mongodb.github.io/mongo-java-driver/ https://mongodb.github.io/mongo-java-driver/3.11/javadoc https://mongodb.github.io/mongo-java-driver/3.11/driver/getting-started/installation/

Veamos una serie de ejemplos para ver el funcionamiento de MongoDB desde JAVA.

7.2 Establecer la conexión

Como se ha comentado en el punto anterior, es necesario añadir el driver de MongoDB para JAVA para poder trabajar. Veamos ahora como establecer una conexión contra MongoDB, recuerda que trabajaremos contra la base de datos en local y que el servicio deberá estar corriendo, para este caso, sin seguridad.

Mongo01Conexion

```
// Log de MongoDB
Logger mongoLogger = Logger.getLogger("org.mongodb.driver");
mongoLogger.setLevel(Level.SEVERE);

// Crear cliente
MongoClient mongoClient = new MongoClient("localhost",27017);

// Seleccionar base de datos
MongoDatabase database = mongoClient.getDatabase("demografia");

// Seleccionar la colección
MongoCollection<Document> collection = database.getCollection("comunidades");

// Mostrar información
System.out.println("BD .........\'" + database.getName() + "\'");
System.out.println("Colección ..\'" + collection.getNamespace() + "\' ");
System.out.println("Número de documentos es " + collection.countDocuments() );
```

En este primer ejemplo podemos ver como se establece una conexión básica, sin ningún tipo de seguridad, seleccionamos la base de datos con la que trabajar y la colección de la que obtendremos la información.

También podemos observar que se utiliza el método **countDocuments()** que nos devuelve el número de documentos que tenemos en la colección.

Importante destacar las líneas de mongoLogger. Nos permiten desactivar el log que muestra el driver de MongoDB, si las comentamos, veremos en todo momento información sobre las acciones que realicemos contra la base de datos.

Conectar con seguridad – Usuarios y Contraseñas

Si tenemos que realizar una conexión utilizando usuario y contraseña debemos utilizar la clase **MongoCredential** para establecer el usuario y contraseña y la base de datos a la que accederá.

Sobre el código anterior modificaremos la conexión de la siguiente forma, para añadir las credenciales.

Mongo01ConexionAuth

```
// Conexión con seguridad.
MongoCredential credenciales = MongoCredential.createCredential(
      "usuadmin",                          // usuario
      "demografia",                        // base de datos
      "mypresidente".toCharArray());       // password
mongoClient = new MongoClient(
      new ServerAddress("localhost", "27017"),
      Arrays.asList(credenciales));
```

El resto del código sería el mismo, ya que lo único que hemos cambiado ha sido la forma de conectar.

Para este ejemplo, MongoDB debería estar funcionando con la seguridad activada, pero nosotros no lo vamos a hacer de momento.

7.3 Obtener datos

Veamos a continuación una de las acciones más importantes sobre las bases de datos, en este caso se verá la obtención de datos como un breve parseo sobre la información obtenida.

Mongo02ReadIterator

```
// Mostramos los datos con while
System.out.println("\nMostrar los datos obtenidos (BSON).");
MongoCursor<Document> cursor = collection.find().iterator();
while (cursor.hasNext()) {
      System.out.println(cursor.next().toJson());
}
```

En este ejemplo se utiliza la clase **MongoCursor** para obtener los documentos de la colección utilizando el método **find()** de la clase **MongoCollection**, y utilizando su método **iterator()** para poder recorrer los documentos.

Veamos ahora, otra forma de obtener los documentos y mostrar el resultado.

Mongo03ReadFor

```
// Mostramos los datos con for
System.out.println("\nMostrar los datos obtenidos (BSON).");
for (Document cursor : collection.find()) {
  System.out.println(cursor.toJson());
}
```

Como vemos en este ejemplo, aprovechamos que obtenemos los documentos mediante **MongoCollection<Document>** para recorrer el array con un bucle for.

Estos dos métodos nos muestran la información en forma Json de la siguiente manera.

```
Mostrar los datos obtenidos (BSON).
{ "_id" : { "$oid" : "57a387e833707ff211d26913" }, "comunidad" : "Comunidad
Valenciana",
"poblacion" : 4953482.0, "provincias" : ["Alicante", "Castellón", "Valencia"] }
{ "_id" : { "$oid" : "57a388d233707ff211d26914" }, "comunidad" : "Región de
Murcia", "poblacion" :
1335792.0, "provincias" : ["Murcia"] }
{ "_id" : { "$oid" : "57a3894233707ff211d26916" }, "comunidad" : "Castilla La
Mancha", "poblacion"
: 2078611.0, "provincias" : ["Toledo", "Ciudad Real", "Guadalajara", "Cuenca",
"Albacete"] }
```

Veamos ahora como darle un formato algo más agradable y más fácil de leer para el usuario, además de comprobar para cada objeto los campos existentes.

Mongo04Read

```
// Mostramos los datos.
System.out.println("Mostrar los datos obtenidos (Parseo)");
System.out.println("===================================");
MongoCursor<Document> cursor = collection.find().iterator();

while (cursor.hasNext()) {
  // DBOject de MongoDB parseado a json
  Document dbObj = Document.parse(cursor.next().toJson());

  // Recuperamos campo comunidad
  if (dbObj.containsKey("comunidad")) {
    System.out.println("Comunidad: " + dbObj.get("comunidad").toString());
  }

  // Recuperamos campo poblacion
  if (dbObj.containsKey("poblacion")) {
    System.out.println("Población: " + dbObj.get("poblacion").toString());
  }

  // Recuperamos campo superficie
  if (dbObj.containsKey("superficie")) {
    System.out.println("Superficie: " + dbObj.get("superficie").toString());
  }

  // Recuperamos el Array de provincias.
  if (dbObj.containsKey("provincias")) {
    ArrayList<String> prov = (ArrayList<String>) dbObj.get("provincias");
    System.out.print("Provincias: ");
    for (Object pr : prov) {
      System.out.print(pr + " ");
    }
    System.out.println();
  }

  System.out.println();

}
```

Como se puede ver, realizamos un parseo de cada documento recogido y utilizamos la clase **Document** para almacenarlo, una vez ahí podemos acceder a cada elemento utilizado el método **get()**.

Fíjate que para recuperar las provincias utilizamos un **ArrayList<String>** con el contenido de las provincias en el DBObject.

Veamos ahora como podemos aplicar **filtros** sencillos a nuestro **find** para seleccionar los datos que necesitemos.

Los filtros se añaden con un objecto JSON como hemos visto en el comando de consola.

Por ejemplo si queremos mostrar las comunidades cuyo nombre comienza por 'Cas' o que tengan una población mayor de 5 millones de habitantes, quedaría:

Mongo05Find

```
// Creamos un filtro
Document filtros = Document.parse(
  "{$or:[  {comunidad:{$regex: '^Cas'}}, {poblacion:{$gt: 5000000}} ]}");

// Mostramos los datos.
System.out.println("Mostrar los datos obtenidos (Parseo)");
System.out.println("====================================");
MongoCursor<Document> cursor = collection.find(filtros).iterator();
```

7.4 Insertar datos

Para insertar datos desde JAVA en MongoDB disponemos de dos métodos que nos ayudarán con esta tarea, insertOne(Document arg0) e insertMany(List<? extends Document> arg0), veamos como utilizar ambas.

En el siguiente ejemplo insertamos un objeto en la comunidad "asignaturas" de la base de datos "instituto":

Mongo06InsertOne

```
// Seleccionamos la BD, si no existe la crea
MongoDatabase db = mongoClient.getDatabase("instituto");

// Seleccionamos la colección, si no existe la crea
MongoCollection<Document> collection = db.getCollection("asignaturas");

/*
* Elimina el contenido de la colección, en este caso,
* para no tener documentos duplicados.
*/
collection.drop();

// Insertar los documentos
System.out.println("Insertando documentos...");

Document doc = new Document();
doc.append("asignatura", "Acceso a datos");
doc.append("profesor", "Pedro Álvarez");
doc.append("horas", 120);
collection.insertOne(doc);

doc = new Document();
doc.append("asignatura", "Desarrollo de interfaces");
doc.append("profesor", "María Rodríguez");
doc.append("horas", 120);
collection.insertOne(doc);

doc = new Document();
doc.append("asignatura", "Programación de servicios y procesos");
doc.append("profesor", "Laura González");
doc.append("horas", 60);
collection.insertOne(doc);

doc = new Document();
doc.append("asignatura", "Programación multimedia y dispositivos móviles");
doc.append("profesor", "Francisco Pérez");
doc.append("horas", 100);
collection.insertOne(doc);
```

En este ejemplo se puede apreciar como tenemos que ir insertando los documentos uno a uno.

Veamos a continuación como insertar varios documentos de una sola vez.

Mongo07InsertMany

```
// Seleccionamos la BD, si no existe la crea
MongoDatabase db = mongoClient.getDatabase("instituto");

// Seleccionamos la colección, si no existe la crea
MongoCollection<Document> collection = db.getCollection("asignaturas");

/*
* Elimina el contenido de la colección, en este caso,
* para no tener documentos duplicados.
*/
collection.drop();

// Insertar los documentos
System.out.println("Insertando documentos...");

// Creamos una lista de documentos
ArrayList<Document> docs = new ArrayList<>();

Document doc = new Document();
doc.append("asignatura", "Acceso a datos");
doc.append("profesor", "Pedro Álvarez");
doc.append("horas", 120);
docs.add(doc);

doc = new Document();
doc.append("asignatura", "Desarrollo de interfaces");
doc.append("profesor", "María Rodríguez");
doc.append("horas", 120);
docs.add(doc);

doc = new Document();
doc.append("asignatura", "Programación de servicios y procesos");
doc.append("profesor", "Laura González");
doc.append("horas", 60);
docs.add(doc);

doc = new Document();
doc.append("asignatura", "Programación multimedia y dispositivos móviles");
doc.append("profesor", "Francisco Pérez");
doc.append("horas", 100);
docs.add(doc);

collection.insertMany(docs);
```

En este ejemplo utilizamos **ArrayList<Document>** para crear un array de documentos al cual iremos añadiendo todos los documentos que sean necesarios, y para terminar, hacemos uso del método **insertMany()** para insertar en la base de datos todo el array de documentos completo.

Veamos a continuación una variante de este último ejemplo.

Mongo08InsertManyClases

```
// Seleccionamos la BD, si no existe la crea
MongoDatabase db = mongoClient.getDatabase("instituto");

// Seleccionamos la colección, si no existe la crea
MongoCollection<Document> collection = db.getCollection("asignaturas");

// Vacía la colección si existe
collection.drop();

// Insertar los documentos
System.out.println("Insertando documentos...");

// Creamos una lista de documentos
ArrayList<Document> docs = new ArrayList<>();

Asignatura asig;
asig = new Asignatura("Acceso a datos", "Pedro Álvarez", 120);
docs.add(asig.toDoc());

asig = new Asignatura("Desarrollo de interfaces", "María Rodríguez", 120);
docs.add(asig.toDoc());

asig = new Asignatura("Programación de servicios y procesos",
                      "Laura González", 60);
docs.add(asig.toDoc());

asig = new Asignatura("Programación multimedia y dispositivos móviles",
                      "Francisco Pérez", 100);
docs.add(asig.toDoc());

collection.insertMany(docs);
```

Para este caso se ha creado la clase **Asignatura.java** y el método **toDoc()**, que se encargará de transformar el objeto Asignatura en un Document.

Asignatura

```
public class Asignatura implements Serializable {

    private String asignatura;
    private String profesor;
    private int horas;

    // Constructores
    ...

    // getters y setters
    ...

    // toString
    ...

    public Document toDoc() {
      return new Document().append("nombre",asignatura)
                      .append("profesor",profesor).append("horas",horas);
    }
}
```

7.5 Actualizar datos

Veamos a continuación el main de la clase **Update.java** para actualizar documentos de uno en uno o varios a la vez.

Mongo09Update

```
// Seleccionamos la BD, si no existe la crea
MongoDatabase db = mongoClient.getDatabase("instituto");

// Seleccionamos la colección, si no existe la crea
MongoCollection<Document> collection = db.getCollection("asignaturas");

// Actualizando los documentos
System.out.println("Actualizando documentos...");

// Actualizamos un sólo documento.
collection.updateOne(
      Filters.eq("nombre", "Desarrollo de interfaces"),
      Updates.set("profesor","Manuel Sánchez")
);

/*
*  Añadimos a todos los documentos, que tengan el campo nombre,
*  el nuevo campo curso.
*/
collection.updateMany(
      Filters.exists("nombre"),
      new Document("$set", new Document("curso", "2ºDAM"))
);
```

Aunque es más sencillo usar la clase estática **Updates**, en el segundo caso, se ha creado un Document para crear el **Document** json **{ $set : { "curso", "2ºDAM" }}** que tiene el mismo efecto.

Filtros

http://mongodb.github.io/mongo-java-driver/3.11/builders/filters/

Updates

http://mongodb.github.io/mongo-java-driver/3.11/builders/updates/

Veamos ahora, como podríamos eliminar de uno o varios documentos un campo en concreto.

```
// Actualizamos los documentos eliminando el campo profesor.
coleccion.updateMany(
      Filters.exists("nombre"),
      Updates.unset("profesor")
);
```

En este caso utilizamos el modificador **unset** de la clase **Updates** para eliminar el campo profesor de todos los documentos.

7.6 Eliminar datos

Para eliminar un documento bastará con la siguiente línea, a la cual puedes aplicar filtros para afinar la operación.

Mongo10DeleteOne

```
// Insertar los documentos
System.out.println("Insertando documento...");

Document doc = new Document();
doc.append("nombre", "Lenguaje de Marcas");
doc.append("profesor", "Noelia Smith");
doc.append("horas", 100);
collection.insertOne(doc);

// Mostramos los datos con for
System.out.println("Mostrar los datos obtenidos (BSON).");
for (Document cursor : collection.find()) {
  System.out.println(cursor.toJson());
}

// Eliminando un documento.
collection.deleteOne(Filters.eq("profesor","Noelia Smith"));

// Mostramos los datos con for
System.out.println("Mostrar los datos obtenidos (BSON).");
for (Document cursor : collection.find()) {
  System.out.println(cursor.toJson());
}
```

También disponemos del método **deleteMany**() de MongoCollection, así como el ya visto en lo ejemplos, **drop**() de la misma clase. Este último eliminará por completo el contenido de una colección.

7.7 Aplicando filtros y orden

Veamos ahora una serie de técnicas para aplicar filtros más avanzados a nuestras consultas, así como métodos de ordenación. Recordad que estas técnicas también se pueden aplicar en los filtros que necesitemos en nuestros **updates** y **deletes**.

Comprobarás que en apartados anteriores se han aplicado algunos filtros y métodos de ordenación, en este apartado vamos a verlos con algo más de detalle. Por ejemplo, para la colección "**comunidades**" de la BD"**demografia**" el filtro del tipo:

```
// (poblacion >= 2000000) AND (poblacion < 5000000)
use demografia
db.comunidades.find({ "$and" : [
      { "poblacion":{$gte : 2000000} } ,
      { "poblacion":{$lt : 5000000} } ]})
```

Utilizando Document con parse

Básicamente es convertir el String JSON a Document:

Mongo11FindFilters

```
// Filtro utilizando Document.parse
System.out.println("Filtro utilizando Document.parse");
System.out.println("================================");
for (Document cur : collection.find(
    Document.parse("{ \"$and\" : [ " +
                        "{ \"poblacion\" : { \"$gte\" : 2000000 } }, "+
                        "{ \"poblacion\" : { \"$lt\" : 5000000 } } "+
                        "] }") )) {
  System.out.println(cur.toJson());
}
System.out.println();
```

Utilizando Document directamente sin parse

Aunque se e podría crear un Document sin parse, complica mucho la sintaxis:

Mongo11FindFilters

```
// Filtro utilizando Document
System.out.println("Filtro utilizando Document sin parse");
System.out.println("====================================");
for (Document cur : collection.find(
                new Document("poblacion",
                    new Document("$gte", 2000000)
                            .append("$lt", 5000000)))
                 ) {
  System.out.println(cur.toJson());
}
```

Esta es otra técnica para filtrar la información, pero en mi opinión, puede ser la más incómoda o tediosa de crear.

Utilizando Filters del API de Java

El API de Java nos facilita la tarea de crear filtros, veamos el ejemplo anterior utilizando la clase Filters del driver.

Mongo11FindFilters

```
// Filtro utilizando Filters
System.out.println("Filtro utilizando Filters");
System.out.println("========================");
for (Document cur : collection.find( and(
                                        gte("poblacion", 3000000),
                                        lt("poblacion", 5000000)) )
                                    ) {
  System.out.println(cur.toJson());
}
```

Utilizando la clase Filters se consigue un filtro más legible, por lo que recomiendo su uso.

En caso de conflicto con los métodos de comparación también se puede indicar la clase estática Filters.

Mongo11FindFilters

```
// Filtro utilizando Filters
System.out.println("Filtro utilizando Filters");
System.out.println("========================");
for (Document cur : collection.find(
                            Filters.and(
                               Filters.gte("poblacion", 3000000),
                               Filters.lt("poblacion", 5000000)) )
                             ) {
  System.out.println(cur.toJson());
}
```

Utilizando Sorts del API de Java

Para ordenar el resultado podremos utilizar la clase **Sorts**.

Mongo12Sorts

```
// Seleccionamos BD y colección
MongoDatabase database = mongoClient.getDatabase("instituto");
MongoCollection<Document> collection = database.getCollection("asignaturas");

// Utilizando Filters y Sorts
Bson filtros = Filters.gt("horas", 90);
Bson orden = Sorts.ascending("profesor");

for (Document cursorLibro : collection.find(filtros).sort(orden)) {
  System.out.println(cursorLibro.toJson());
}
```

Ejemplo avanzado usando Filters y Sorts

Vamos a realizar una aplicación que procese varias colecciones ordenando el resultado.

Utlizando la BD "**gimnasio**", mostrar los datos de la colección "**clases**" mostrando los datos del "**monitor**" que imparte la clase y de los "**clientes**" que asisten:

Mongo13Gimnasio

```
// Conexión a MongoDB, Database y Colección
MongoClient mongoClient = new MongoClient("localhost", 27017);
MongoDatabase database = mongoClient.getDatabase("gimnasio");
MongoCollection<Document> colClases = database.getCollection("clases");
MongoCollection<Document> colMonitores = database.getCollection("monitores");
MongoCollection<Document> colClientes = database.getCollection("clientes");

// Mostramos los datos
System.out.println("Clases del GIMNASIO");
System.out.println("==================================");
MongoCursor<Document> cursor =
                colClases.find().sort(Sorts.descending("nombre")).iterator();

while (cursor.hasNext()) {
  // DBOject de MongoDB parseado a json
  Document oClase = Document.parse(cursor.next().toJson());

  System.out.println("Código.........: " + oClase.get("cod_clase").toString());
  System.out.println("Nombre.........: " + oClase.get("nombre").toString());
  System.out.println("Horas semanales: " + oClase.get("horas_sem").toString());

  // Monitor
  Document oMonitor = colMonitores.find(
      Filters.eq("id_monitor",
          oClase.get("impartida_por").toString() )
        ).iterator().next();

  System.out.println("Monitor........: "
        + oMonitor.get("nombre") + " " + oMonitor.get("apellidos") );

  // Clientes
  ArrayList<String> asisten = (ArrayList<String>) oClase.get("asisten");
  System.out.println("Asisten: ");
  for (Object cli : asisten) {
    Document oCliente = colClientes.find(Filters.eq("dni",cli))
                                        .iterator().next();
    System.out.println(" -> "
        + oCliente.get("nombre") + " "
        + oCliente.get("apellidos")
        + "(" + oCliente.get("telefono") + ")" );
  }
  System.out.println("==================================");
  System.out.println();
}
```

En este último código de ejemplo se combina una ordenación con un filtro simple tanto para monitores como para clientes, y todo mediante las clases Filters y Sorts.

Filters
http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/client/model/Filters.html

Sorts
http://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/client/model/Sorts.html

Parar profundizar más en MongoDB
http://gpd.sip.ucm.es/rafa/docencia/nosql/ http://gpd.sip.ucm.es/rafa/docencia/nosql/shell.html http://gpd.sip.ucm.es/rafa/docencia/nosql/Agregando.html http://gpd.sip.ucm.es/rafa/docencia/nosql/seguridad.html